DKRZ
DEUTSCHES
KLIMARECHENZENTRUM

HLRE-3 MISTRAL
User's Manual

Support:
beratung@dkrz.de

2015-08-25

Contents

1 Cluster Information **4**
- 1.1 Introduction . . . 4
- 1.2 Cluster Nodes . . . 4
- 1.3 Data Management - Filesystems . . . 5
  - 1.3.1 Data Migration from Blizzard . . . 6
- 1.4 Access to the Cluster . . . 6
  - 1.4.1 Login . . . 7
  - 1.4.2 Password . . . 7
  - 1.4.3 Login Shell . . . 7

2 Software Environment **8**
- 2.1 Modules . . . 8
  - 2.1.1 Modules Available . . . 8
  - 2.1.2 Using the Module Command . . . 9
- 2.2 Compiler and MPI . . . 9
  - 2.2.1 Compilation Examples . . . 10
  - 2.2.2 Recommendations . . . 11

3 Batch System - SLURM **13**
- 3.1 SLURM Overview . . . 13
- 3.2 SLURM Partitions . . . 14
- 3.3 Job Limits - QoS . . . 15
- 3.4 Priorities and Accounting . . . 16
- 3.5 Job Environment . . . 16

4 SLURM Usage **17**
- 4.1 SLURM Command Overview . . . 17
- 4.2 Allocation Commands . . . 18
  - 4.2.1 Interactive Jobs . . . 18
  - 4.2.2 Spawning Command . . . 19
  - 4.2.3 Batch Jobs . . . 19
- 4.3 Job Script Examples . . . 22
- 4.4 Advanced SLURM Features . . . 27
  - 4.4.1 Hyper-Threading (HT) . . . 27
  - 4.4.2 Process and Thread Binding . . . 28
  - 4.4.3 MPMD . . . 31
  - 4.4.4 Frequency Scaling . . . 32
  - 4.4.5 Job Steps . . . 32
  - 4.4.6 Dependency Chains . . . 33
  - 4.4.7 Job Arrays . . . 33
- 4.5 SLURM Command Examples . . . 35

4.5.1 Query Commands 35
4.5.2 Job Control 36
4.5.3 Accounting Commands 37

Chapter 1

Cluster Information

1.1 Introduction

MISTRAL, the High Performace Computing system for Earth system research (HLRE-3), is DKRZ's first petascale supercomputer by Atos. The phase 1 configuration with a peak performance of roughly 1.5 PetaFLOPS consists of more than 1,500 compute nodes, 36,000 compute cores, 110 Terabytes of memory, and 20 Petabytes of disk. To access MISTRAL you need to be a member in at least one active HLRE project, to have a valid user account, and to accept DKRZ's "Guidelines for the use of information-processing systems of the Deutsches Klimarechenzentrum GmbH (DKRZ)".

1.2 Cluster Nodes

The MISTRAL cluster in phase 1 consists of more than 1500 nodes. The compute nodes are housed in bullx B700 DLC (Direct Liquid Cooling) blade systems with two nodes forming one blade. Each node has two sockets, equipped with an Intel Xeon E5-2680 v3 12-core processor (Haswell) each. Thus, 24 physical cores per node are available. Due to active Hyper-Threading, the operating system recognizes two logical cores per physical core. The processor clock-rate is 2.5 GHz. The peak performance of the system is about 1.5 PFLOPS/s. The aggregated main memory is 115 TB. The parallel file system Lustre provides 20 PB of usable disk space.

Four kinds of nodes are available to users: 8 login nodes, 1496 compute nodes for running scientific models, 48 nodes for interactive use and pre- and postprocessing of data, and 12 visualisation nodes. See Table 1.1 for a listing of the specifics of different node types.

type (nodes)	hostname	CPU	GPUs	memory
login (8)	mlogin[100-107]	2x12 core Intel Haswell @ 2.5GHz	none	256 GB
compute (1386)	m[10000-11367], m[11404-11421]	2x12 core Intel Haswell @ 2.5GHz	none	64 GB
compute (110)	m[11368-11403], m11422, m11431, m[11440-11511]	2x12 core Intel Haswell @ 2.5GHz	none	128 GB
pre/post (48)	m[11512-11559]	2x12 core Intel Haswell @ 2.5GHz	none	256 GB
viz (12)	mg[100-111]	2x12 core Intel Haswell @ 2.5GHz	Nvidia Tesla K80 2x GK110BGL	256 GB

Table 1.1: MISTRAL node configuration

The Operating System on the MISTRAL cluster is Red Hat Enterprise Linux release 6.4 (Santiago). All compute, pre-/postprocessing, and visualization nodes are integrated in one FDR InfinBand (IB) fabric with three Mellanox SX6536 director switches and fat tree topology with a blocking factor of 1:2:2. The measured bandwidth between two arbitrary compute nodes is 5.9 GByte/s with a latency of 2.7 μs.

1.3 Data Management - Filesystems

On MISTRAL we provide the Lustre parallel filesystem version 2.5. Users have access to three different storage spaces: HOME, WORK, and SCRATCH. Each storage area has a specific purpose as described below.

HOME is the file system where users' sessions start upon login to MISTRAL. It is backed up and should be used to store shell setup files, source codes, scripts, and important files.

WORK is a project space available through the allocations process and shared between all users of a project. It provides disk space for large amounts of data, but it is not backed up. It can be used e.g. for writing raw model output and processing of data that is accessible to all project members.

SCRATCH is provided for temporary storage and processing of large data. To prevent the file system from overflowing, old data is automatically deleted. The granted retention period is 14 days.

All file systems are available on all nodes (login and compute), so you can use them during interactive sessions and in batch jobs. The table below provides further details on available file systems.

File System	HOME	WORK	SCRATCH
path	/pf/[a,b,g,k,m,u]/<userid>	/work/<project>	/scratch/[a,b,g,k,m,u]/<userid>
envVar	$HOME		
description	• Assigned to user account • Storage of personal settings files, source codes and scripts	• Assigned to project account • Interim storage of output from running applications and frequently accessed data	• Assigned to user account • Temporary storage and processing of large data sets
quota	24 GB	according to annual project allocation	15 TB
backup	yes, please contact DKRZ user's consultancy to restore files deleted by mistake	**no**	**no**
automatic data deletion	no	no	**yes**
data life time	until user account deletion	1 month after project expiration	14 days since the last file access

Table 1.2: MISTRAL file system configuration

1.3.1 Data Migration from Blizzard

- The users' home directories from blizzard have been copied to mistral under

  `/mnt/lustre01/rsync/pf`

  The last copy was made on August 1st 2015. Please copy the files you need to your actual home directory on mistral `/pf/[a,b,g,k,m,u]/<userid>`

- `/pool/data` is mirrored from blizzard to the same directory on mistral

- The project directories on `/work` have been copied to mistral under

  `/mnt/lustre01/rsync/work`

  The last copy was made August 1st 2015. Please move all data you want to keep to your actual project directory `/work/<projectid>`

1.4 Access to the Cluster

The High Performance Computing system MISTRAL can be only accessed via Secure Shell (SSH) network protocol. For file transfer between different hosts SSH provides SCP and SFTP.

1.4.1 Login

You can log into MISTRAL with the following ssh command, replacing <userid> by your username:

```
bash$ ssh <userid>@mistral.dkrz.de
```

After having logged into MISTRAL, you will find yourself on one of the eight login nodes: mlogin100-mlogin107. The login nodes serve as front end to the compute nodes of the HPC cluster. They are intended for file editing and compilation of source code, as well as for submitting, monitoring and cancelling of batch jobs. They can also be used for none time- and memory-intensive serial processing tasks. The routine data analysis and visualization, however, have to be performed on pre-/post-processing nodes or on visualization servers. For interactive testing and debugging of parallel programs you can use SLURM `salloc` command to allocate the required number of nodes.

1.4.2 Password

All DKRZ systems are managed by the LDAP protocol. The password can be changed through DKRZ online services. A user defined password must contain at least eight non blank characters and must be a combination of upper and lower-case letters, numbers and special characters. In case you do not remember your password please contact DKRZ user's consultancy. Members of MPI and UniHH/CEN should contact CIS/CEN-IT.

1.4.3 Login Shell

The default login shell for new DKRZ users is bash. You can change your login shell to tcsh or ksh using the DKRZ online services. The settings you would like to use every time you log in can be put into special shell setup files. A login bash shell looks for `.bash_profile`, `.bash_login` or `.profile` in your home directory and executes commands from the first file found. A non-login bash shell or bash subshell reads `.bashrc` file. Tcsh always reads and executes `.cshrc` file. If tcsh is invoked as login shell, the file `.login` is sourced additionally. The typical tasks and settings that can be put in shell setup files are for example:

- Creation of a custom prompt
- Modification of search path for external commands and programs
- Definition of environment variables needed by programs or scripts
- Definition of aliases
- Execution of commands (e.g. '`module load <modname>/<version>`')

Chapter 2

Software Environment

2.1 Modules

To cover the software needs of DKRZ users and to maintain different software versions, the DKRZ uses the module environment. Loading a module adapts your environment variables to give you access to a specific set of software and its dependencies. The modules are not organized hierarchically but have internal consistency checks for dependencies and can uniquely be identified by naming convention `<modname>/<modversion>`. Optionally, the version of the compiler that was used to build the software is also encoded in the name (for example all modules built with the same Intel compiler version are labelled with e.g. `*-intel14`).

2.1.1 Modules Available

Table 2.1 provides a quick reference to some module categories. The list of available modules will steadily grow to cover the (general) software needs of DKRZ users. A complete list is dynamically updated whenever new software was built and can be found at https://www.dkrz.de/Nutzerportal-en/doku/mistral/softwarelist

type	modules available
compiler	**intel**: Intel compilers with frontends for C, C++ and Fortran **gcc**: Gnu compiler suite **nag**: NAG compiler
MPI	**intelmpi**: Intel MPI **bullxmpi**: Bullx-MPI with/without mellanox libraries **mvapich2**: MVAPICH2 (an MPI-3 implementation) **openmpi**: Open MPI
tools	**allinea-forge**: Allinea DDT debugger and MAP profiler **cdo**: command line Operators to manipulate and analyse Climate and NWP model Data **ncl**: NCAR Command Language **ncview**: visual browser for netCDF format files **python**: Python

Table 2.1: MISTRAL module overview

2.1.2 Using the Module Command

Users can load, unload and query modules through the module command. The most important module sub-commands are listed in the table below.

command	description
module avail	Shows the list of all available modules
module show <modname>/<version>	Shows environment changes the modulefile <modname>/<version> will cause if loaded
module add <modname>/<version>	Loads a specific module. Default version is loaded if the version is not given
module list	Lists all modules currently loaded
module rm <modname>/<version>	Unloads a module
module purge	Unloads all modules
module switch <modname>/<version1> <modname>/<version2>	Replaces one module with another

Table 2.2: module command overview

For all details of the module command please refer to the man page or execute '`module --help`'. To use the module command in a script you can source one of the following files in your script before any invocation of the module command:

```
# in bash or ksh script
source /sw/rhel6-x64/etc/profile.mistral

# in tcsh or csh script
source /sw/rhel6-x64/etc/csh.mistral
```

The '`module avail`' command provides up-to-date information on installed software and versions.

2.2 Compiler and MPI

On MISTRAL we provide the Intel, GCC (GNU Compiler Collection), and NAG compilers and several Message Passing Interface (MPI) implementations: Bullx MPI with and without Mellanox MXM and FCA tools, Intel MPI, MVAPICH2, and OpenMPI. No compilers and MPIs are loaded by default.

For most applications we recommend to use the Intel compilers and Bullx MPI library with Mellanox tools to achieve the optimal performance on MISTRAL. For some applications running on small number of nodes slightly better performance might be achieved with the Intel compilers and Intel MPI.

Compiler and an appropriate MPI library can be selected by loading the corresponding module files, for example:

```
# Use the default versions of Intel compiler and Bullx MPI with Mellanox MXM + FCA tools
$ module load intel mxm fca bullxmpi_mlx

# Use the default versions of Intel compiler and Intel MPI
$ module load intel intelmpi
```

The following table shows the names of the MPI wrapper procedures for the Intel compilers as well as the names of compilers themselves. The wrappers build up the MPI environment for your compilation task, such that we recommend the use of the wrappers instead of the compilers themselves.

language	**compiler**	**Intel MPI Wrapper**	**bullx MPI Wrapper**
Fortran 90/95/2003	ifort	mpiifort	mpif90
Fortran 77	ifort	mpiifort	mpif77
C++	icpc	mpiicpc	mpic++
C	icc	mpiicc	mpicc

Table 2.3: MPI compiler wrapper overview for Intel compiler

The table below lists some useful compiler options that are commonly used for the Intel compiler. For further information please refer to the man pages of the compiler or the comprehensive documentation on the Intel website https://software.intel.com/en-us/intel-software-technical-documentation.

option	**description**
−openmp	Enables the parallelizer to generate multi-threaded code based on the OpenMP directives
−g	Creates debugging information in the object files. This is necessary if you want to debug your program
−O[0−3]	Sets the optimization level
−L<library path>	A path can be given in which the linker searches for libraries
−D	Defines a macro
−U	Undefines a macro
−I<include path>	Allows to add further directories to the include file search path
−sox	Stores useful information like compiler version, options used etc. in the executable
−ipo	Inter-procedural optimization
−xAVX or −xCORE-AVX2	Indicates the processor for which code is created
−help	Gives a long list of quite a big amount of options

Table 2.4: Intel compiler options

2.2.1 Compilation Examples

Compile a hybrid MPI/OpenMP program using the Intel Fortran compiler and Bullx MPI with MXM and FCA:

```
$ module add intel mxm fca bullxmpi_mlx
$ mpif90 −openmp −O2 −xCORE−AVX2 −fp−model source −o mpi_omp_prog program.f90
```

Compile an MPI program in Fortran using Intel Fortran compiler and Intel MPI:

```
$ module add intel intelmpi
$ mpiifort −O2 −xCORE−AVX2 −fp−model source −o mpi_prog program.f90
```

2.2.2 Recommendations

Intel Compiler

Using the compiler option `-xCORE-AVX2` resp. `-xHost` causes the Intel compiler to use full AVX2 support/vectorization (with FMA instructions) which might result in binaries that do not produce MPI decomposition independent results. Switching to `-xAVX` should solve this issue but result in up to 15% slower runtime.

MPI

The bullx-MPI was used throughout for the benchmarks of the HLRE-3 procurement. From BULL/ATOS point of view, a good environment will be to use `bullxMPI_mlx` with MXM, i.e. load the specific environment before compiling

```
$ module add intel mxm/3.3.3002 fca/2.5.2379 bullxmpi_mlx/bullxmpi_mlx- 1.2.8.3
$ mpif90 -O2 -xCORE-AVX2 -o mpi_prog program.f90
```

One must respect the order of loading the modules: compiler, MXM/FCA and afterwards bullx MPI. If the MXM/FCA environment is not loaded, one will use the bullx MPI without MXM and FCA tools.

In order to use the MXM (Mellanox Messaging) to accelerate the underlying send/receive (or put/get) messages, the following variables have to be used

```
export OMPI_MCA_pml= cm
export OMPI_MCA_mtl= mxm
export MXM_RDMA_PORTS= mlx5_0:1
```

Furthermore, FCA (Fabric Collectives Accelerations) accelerates the underlying collective operations used by the MPI/PGAS languages. To use FCA, one must specify the following variables

```
export OMPI_MCA_coll= ^ghc
export OMPI_MCA_coll_fca_priority= 95
export OMPI_MCA_coll_fca_enable= 1
```

You will find the bullxMPI documentation by Atos at
https://www.dkrz.de/Nutzerportal-en/doku/mistral/manuals.

Libraries

There is no module to set NetCDF paths for the user. If you need to specify such paths in Makefiles or similar, please use the nc-config and nf-config tool to get the needed compiler flags and libraries, e.g.

```
# Get paths to netCDF include files
$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/bin/nc-config --cflags

-I/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/include \
-I/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/include \
-I/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/include

# Get options needed to link a C program to netCDF
$ /sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/bin/nc-config --libs

-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
```

```
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib -lnetcdf


# Get paths to Fortran netCDF include files
$ /sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/bin/nf-config --fflags

-I/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/include

# Get options needed to link a Fortran program to netCDF
$ /sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/bin/nf-config --flibs

-L/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/lib -lnetcdff \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_fortran-4.4.2-intel14/lib \
-L/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/netcdf/netcdf_c-4.3.2-gcc48/lib \
-L/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/hdf5/hdf5-1.8.14-threadsafe-gcc48/lib \
-L/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/lib \
-Wl,-rpath,/sw/rhel6-x64/sys/libaec-0.3.2-gcc48/lib \
-lnetcdf -lhdf5_hl -lhdf5 -lsz -lcurl -lz
```

Chapter 3

Batch System - SLURM

3.1 SLURM Overview

SLURM is the Batch System (Workload Manager) used on MISTRAL cluster. SLURM (Simple Linux Utility for Resource Management) is a free open-source resource manager and scheduler. It is a modern, extensible batch system that is widely deployed around the world on clusters of various sizes. A SLURM installation consists of several programs/user commands and daemons which are shown in Table 3.1 and Figure 3.1.

daemon	**description**
control daemon (**slurmctld**)	responsible for monitoring of available resources and scheduling of batch jobs, it is running on admin nodes as HA resource
database daemon (**slurmdbd**)	accessing and managing the MySQL database which stores all the information about users, jobs and accounting data
slurm daemon (**slurmd**)	functionality of the batch system and resource management, it is running on each compute node
step daemon (**slurmstepd**)	a job step manager spawned by slurmd to guide the user processes

Table 3.1: Overview on SLURM components

SLURM manages the compute, pre-/post-processing and visualisation nodes as its main resource of the cluster. Several nodes are grouped together into partitions, which might overlap, i.e. one node might be contained in several partitions. Compared to LoadLeveler on BLIZZARD, partitions are the equivalent of classes, hence partitions are the main concept for users to start jobs on the MISTRAL cluster.

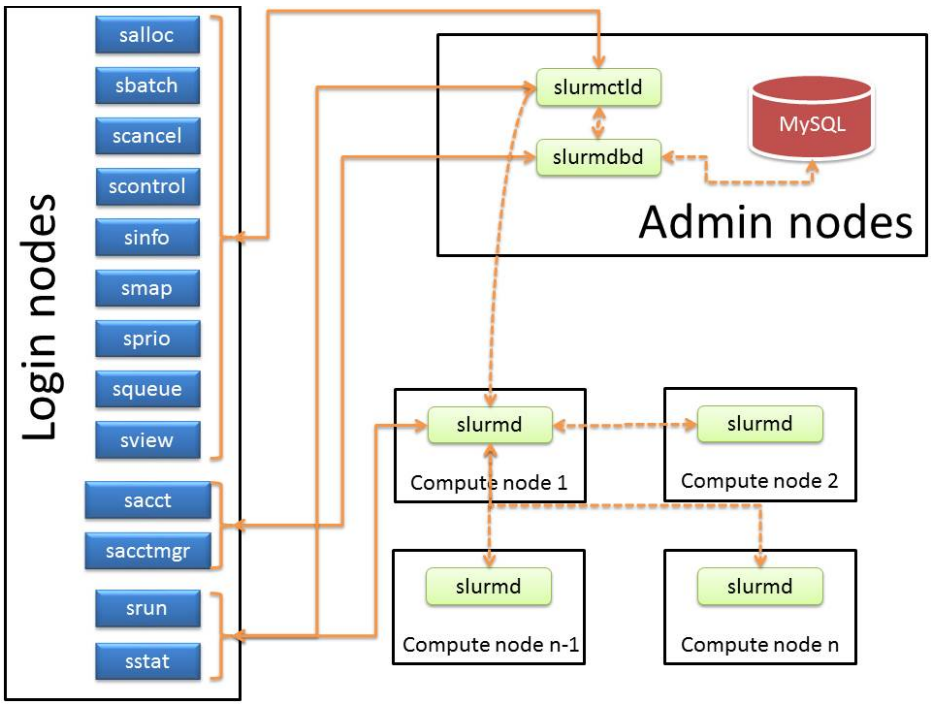

Figure 3.1: SLURM daemons and their interaction

3.2 SLURM Partitions

In SLURM multiple nodes can be grouped into **partitions** which are sets of nodes with associated limits for wall-clock time, job size, etc. These limits are hard-limits for the jobs and can not be overruled.

Jobs are the allocations of resources by the users in order to execute tasks on the cluster for a specified period of time. Furthermore, the concept of **jobsteps** is used by SLURM to describe a set of different tasks within the job. One can imagine jobsteps as smaller allocations or jobs within the job, which can be executed sequentially or in parallel during the main job allocation.

The SLURM `sinfo` command lists all partitions and nodes managed by SLURM on MISTRAL as well as provides general information about the current nodes' status:

```
bash$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up    8:00:00     13  down* m[10000,10278,10286,10438,
compute*     up    8:00:00    168   alloc m[10036-10045,10108-10125,
compute*     up    8:00:00   1315    idle m[10001-10035,10046-10107,
prepost      up    4:00:00      3 drain* m[11518,11532,11554]
prepost      up    4:00:00     45    idle m[11512-11517,11519-11531,
shared       up 7-00:00:00      1  down* m10000
shared       up 7-00:00:00     28   alloc m[10036-10045,11314-11331]
shared       up 7-00:00:00     71    idle m[10001-10035,10046-10049,
gpu          up    4:00:00      1  down* mg102
gpu          up    4:00:00     11    idle mg[100-101,103-111]
```

For detailed information about all available partitions and their limits use the SLURM `scontrol` command as follows:

```
$ scontrol show partition
```

The following four partitions are currently defined on MISTRAL:

compute This is default partition consisting of 1496 compute nodes and intended for running parallel scientific applications. The compute nodes allocated for a job are used exclusively and can not be shared with other jobs.

shared This partition is defined on 100 nodes and can be used to run small jobs not requiring a whole node for the execution, so that one compute node can be shared between different jobs. The partition is dedicated for execution of shared memory applications parallelized with OpenMP or pthreads as well as for serial and parallel data processing jobs.

prepost The prepost partition is made up of 48 large-memory nodes. It is dedicated for memory intensive data processing jobs. Furthermore, interactive usage of nodes is permitted on this partition. If over-subscription is explicitly requested by the user using the "–share" option on job submission, resources can be shared with other jobs.

gpu The 12 nodes in this partition are additionally equipped with Nvidia Tesla K80 GPUs and can be used for 3-dimensional data visualization or execution of applications ported to GPUs. The nodes in this partition will replace Halo cluster in the future.

The limits configured for different partitions are listed in the table below.

partition	**compute**	**prepost**	**shared**	**gpu**
MaxNodes	512	2	1	1
MaxTime	8 hours	4 hours	7 days	4 hours
Shared	exclusive	yes	yes	exclusive
MaxMemPerCPU	nodelimit	5 GByte	2.5 GByte	5 GByte

Table 3.2: Overview on SLURM partitions for MISTRAL

3.3 Job Limits - QoS

As stated above, the partitions have several hard-limits that put an upper limit for the jobs on the wall-clock or other constraints. However, the actual job limits are enforced by the limits specified in both partitions and so called Quality-of-Services (QoS), which means that using a special QoS the user might weaken the partition limits.

These QoSs play an important role to define the job priorities. By defining some QoSs the possible priorities can be modified in order to e.g. enable earlier starttime of jobs. Following we present the current list with the configured Quality-of-Services. If users have any demand for creating new QoS, we kindly ask to contact us.

QoS	**express**
description	higher priority
limits	4 nodes, 20 min wallclock

Table 3.3: Overview on SLURM QoS for MISTRAL

3.4 Priorities and Accounting

The main policies concerning the batch model and accounting that are applied on MISTRAL are also defined via SLURM.

- SLURM schedules the jobs according to their priorities. The jobs with the highest priorities will be scheduled next.
- Usage of backfilling scheduling algorithm: the SLURM scheduler checks the queue and may schedule jobs with lower priorities that can fit in the gap created by freeing resources for the next highest priority jobs.
- For each project a SLURM account is created where the users belong to. Each user might use the contingent from several projects that he belongs to.
- Users can submit jobs even when granted shares are already used - this result in a low priority, but the job might start when the system is empty.

SLURM has a very simple and well defined priority mechanism that allows to define different weighting models. The actual priority is based on five factors to calculate the job priorities: Age, Fairshare, Job-Size, Partition and QoS

$$\begin{aligned} Job_priority = & (PriorityWeightAge) * (age_factor) + \\ & (PriorityWeightFairshare) * (fairshare_factor) + \\ & (PriorityWeightJobSize) * (job_size_factor) + \\ & (PriorityWeightPartition) * (partition_factor) + \\ & (PriorityWeightQOS) * (QOS_factor) \end{aligned} \tag{3.1}$$

For each factor a weight is defined to balance the job-priority equation:

- *WeightQOS*: 10000
- *WeightAge*: 1000
- *WeightJobSize*: TODO
- *WeightFairshare*: 100000
- *WeightPartition*: 10000

3.5 Job Environment

On the compute nodes the whole shell environment is passed to the jobs during submission. With some options of the allocation commands (like `--export` for the `sbatch` command), users can change this default behaviour. The users can load modules and prepare the desired environment before job submission, and then this environment will be passed to the jobs that will be submitted. Of course, a good practice is to include module commands inside the job-scripts, in order to have full control of the environment of the jobs.

Chapter 4

SLURM Usage

This chapter serves as an overview of user commands provided by SLURM and how users should use the SLURM batch system in order to run jobs on MISTRAL. For a comparison to LoadLeveler commands see http://slurm.schedmd.com/rosetta.pdf or read the more detailed description of each command's manpage. A concise cheat sheet for SLURM can be downloaded here: http://slurm.schedmd.com/pdfs/summary.pdf

4.1 SLURM Command Overview

SLURM offers a variety of user commands for all the necessary actions concerning the jobs. With these commands the users have a rich interface to allocate resources, query job status, control jobs, manage accounting information and to simplify their work with some utility commands. For examples how to use these command, see Chapter 4.5.

sinfo show information about all partitions and nodes managed by SLURM as well as about general system state. It has a wide variety of filtering, sorting, and formatting options.

squeue query the list of pending and running jobs. By default it reports the list of pending jobs sorted by priority and the list of running jobs sorted separately according to the job priority. The most relevant job states are running (R), pending (PD), completing (CG), completed (CD) and cancelled (CA). The TIME field shows the actual job execution time. The NODELIST (REASON) field indicates on which nodes the job is running or the reason why the job is pending. Typical reasons for pending jobs are waiting for resources to become available (Resources) and queuing behind a job with higher priority (Priority).

sbatch submit a batch script. The script will be executed on the first node of the allocation. The working directory coincides with the working directory of the sbatch directory. Within the script one or multiple srun commands can be used to create job steps and execute parallel applications.

scancel cancel a pending or running job or job step. It can also be used to send an arbitrary signal to all processes associated with a running job or job step.

salloc request interactive jobs/allocations. When the job is started a shell (or other program specified on the command line) is started on the submission host (login node). From this shell you should use srun to interactively start a parallel applications. The allocation is released when the user exits the shell.

srun initiate parallel job steps within a job or start an interactive job.

scontrol (primarily used by the administrators) provides some functionality for the users to manage jobs or get some information about the system configuration such as nodes, partitions, jobs, and configurations.

sprio query job priorities.

sshare retrieve fair-share information for each account the user belongs to.

sstat query status information related to CPU, task, node, RSS and virtual memory about a running job.

sacct retrieve accounting information about jobs and job steps. For completed jobs sacct queries the accounting database.

4.2 Allocation Commands

A job allocation, i.e. a request on compute resources, can be created using the SLURM `salloc`, `sbatch` or `srun` command.

The usual way to allocate resources and execute a job on MISTRAL is to write a batch script and submit it to SLURM with the `sbatch` command - see section 4.2.3 for details. Alternatively, an interactive allocation can be used via the `salloc` command or a parallel job can directly be started with the `srun` command.

4.2.1 Interactive Jobs

Interactive sessions can be allocated using the `salloc` command. The following command for example will allocate 2 nodes for 30 minutes:

```
$ salloc --nodes=2 --time=00:30:00 --account=x12345
```

Once an allocation has been made, the `salloc` command will start a bash shell on the **login node** where the submission was done. After a successful allocation the users can execute `srun` from that shell to spawn interactively their applications. For example:

```
$ srun --ntasks=4 --ntasks-per-node=2 --cpus-per-task=4 ./my_code
```

The interactive session is terminated by exiting the shell. In order to run commands directly on the allocated compute nodes, the user has to use ssh to connect to the desired nodes. For example:

```
$ salloc --nodes=2 --time=00:30:00 --account=x12345
salloc: Granted job allocation 13258

$ squeue -j 13258
JOBID PARTITION   NAME     USER ST   TIME  NODES NODELIST(REASON)
13258   compute   bash  x123456  R   0:11      2 m[10001-10002]

$ hostname # we are still on the login node
mlogin103

$ ssh m10001

user@m10001:~$ hostname
```

```
m10001

user@m10001:~$ exit
logout
Connection to m10001 closed.

$ exit # we need to exit in order to release the allocation
salloc: Relinquishing job allocation 13258
salloc: Job allocation 13258 has been revoked.
```

4.2.2 Spawning Command

With `srun` the users can spawn any kind of application, process or task inside a job allocation or directly start executing a parallel job (and indirectly ask SLURM to create the appropriate allocation). It can be a shell command, any single-/multi-threaded executable in binary or script format, MPI application or hybrid application with MPI and OpenMP. When no allocation options are defined with `srun` command the options from `sbatch` or `salloc` are inherited.

`srun` should preferably be used either

1. inside a job script submitted by `sbatch` - see 4.2.3.
2. or after calling `salloc`.

The allocation options of `srun` for the job-steps are (almost) the same as for `sbatch` and `salloc` (please see the table in section 4.2.3 for some allocation options).

Examples:

Spawn 48 tasks on 2 nodes (24 tasks per node) for 30 minutes:

```
$ srun -N 2 -n 48 -t 30 -A xy1234 ./my_small_test_job
```

You will have to specify the account to be used for this job in the same manner as for `salloc` and `sbatch`.

4.2.3 Batch Jobs

Users submit batch applications using the `sbatch` command. The batch script is usually a shell script consisting of two parts: resources requests and job steps. Resources requests are for example number of nodes needed to execute the job, number of tasks, time duration of the job etc. Job steps are user's tasks that must be executed. The resources requests and other SLURM submission options are prefixed by '`#SBATCH`' and must precede any executable commands in the batch script. For example:

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --account=xz0123
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00

# Begin of section with executable commands
set -e
ls -l
srun ./my_program
```

The script itself is regarded by SLURM as the first job step and is (serially) executed on the first compute node in the job allocation. To execute parallel MPI tasks users call `srun` within their script. Thereby, a new job step is initiated. It is possible to execute parallel programs in the form of job steps in any configuration within the job allocation. This means, a job step can use all allocated resources or several job steps (created via multiple `srun` calls) can use a subset of allocated resources.

The following table describes the most common or required allocation options that can be defined in a batch script:

#SBATCH option	default value	description
`--nodes=<number>` `-N <number>`	1	Number of nodes for the allocation
`--ntasks=<number>` `-n <number>`	1	Number of tasks (MPI processes). Can be omitted if `--nodes` and `--ntasks-per-node` are given
`--ntasks-per-node=<number>`	1	Number of tasks per node. If keyword omitted the default value is used, but there are still 48 CPUs available per node for current allocation (if not shared)
`--cpus-per-task=<number>` `-c <number>`	1	Number of threads (logical cores) per task. Used mainly for OpenMP or hybrid jobs
`--output=<path>/<file pattern>` `-o <path>/<file pattern>`	`slurm-%j.out`	Standard output file
`--error=<path>/<file pattern>` `-e <path>/<file pattern>`	`slurm-%j.out`	Standard error file
`--time=<walltime>` `-t <walltime>`	partition dep.	Requested walltime limit for the job
`--partition=<name>` `-p <name>`	compute	Partition to run the job
`--mail-user=<email>`	username	Email address for notifications
`--mail-type=<mode>`	NONE	Event types for email notifications. Possible values are NONE, BEGIN, END, FAIL, REQUEUE, ALL, TIMELIMIT
`--job-name=<jobname>` `-J <jobname>`	job script's name	Job name
`--account=<project>` `-A <project>`	none	Project that should be charged
`--requeue` `--no-requeue`	requeue	Specifies whether the batch job should be requeued after a node failure. When a job is requeued, the batch script is initiated from its beginning!

Table 4.1: SLURM sbatch options

Multiple `srun` calls can be placed in a single batch script. Options such as `--nodes`, `--ntasks` and `--ntasks-per-node` are inherited from the `sbatch` arguments but can be

overwritten for each `srun` invocation.

The complete list of parameters can be inquired from the `sbatch` man page:

```
$ man sbatch
```

As already mentioned above the batch script is submitted using the SLURM `sbatch` command:

```
$ sbatch [OPTIONS] <jobscript>
```

On success, `sbatch` writes the job ID to standard output. Options provided on command line supersede the same options defined in the batch script.

Remember the difference between options for selection, allocation and distribution in SLURM. Selection and allocation works with **sbatch**, but task distribution and binding should directly be specified with **srun** (within an sbatch-script). The following steps give an overview, for details see the further documentation below.

1. Resource Selection, e.g.
   - `#SBATCH --nodes=2`
   - `#SBATCH --sockets-per-node=2`
   - `#SBATCH --cores-per-socket=12`
2. Resource Allocation, e.g.
   - `#SBATCH --ntasks=12`
   - `#SBATCH --ntasks-per-node=6`
   - `#SBATCH --ntasks-per-socket=3`
3. Start the application relying on the sbatch options only. Task binding and distribution with `srun`, e.g.

   ```
   srun --cpu_bind=cores --distribution=block:cyclic <my_binary>
   ```

4. Start the application using only parts of the allocated resources, one needs to give again all relevant allocation options to `srun` (like `--ntasks` or `--ntasks-per-node`), e.g.

   ```
   srun --ntasks=2 --ntasks-per-node=1 --cpu_bind=cores \
   --distribution=block:cyclic <my_binary>
   ```

All environment variables set at the time of submission are propagated to the SLURM jobs. With some options of the allocation commands (like `--export` for sbatch or srun), users can change this default behaviour. The users can load modules and prepare the desired environment before job submission, and then this environment will be passed to the jobs that will be submitted. Of course, a good practice is to include module commands in job scripts, in order to have full control of the environment of the jobs.

NOTE: on the MISTRAL cluster setting of `-A` resp. `--account` is necessary to submit a job, otherwise submission will be rejected. You can query the accounts for which job submission is allowed using the command:

```
bash$ sacctmgr list assoc format=account,qos,MaxJobs user=$USER
```

4.3 Job Script Examples

Serial job

```
#!/bin/bash

#SBATCH --job-name=my_job       # Specify job name
#SBATCH --partition=shared      # Specify partition name
#SBATCH --ntasks=1              # Specify max. number of tasks
                                # to be invoked
#SBATCH --mem-per-cpu=<MB>      # Specify  real memory required per CPU
#SBATCH --time=00:30:00         # Set a limit on the total run time
#SBATCH --mail-type=FAIL        # Notify user by email in case of
                                # job failure
#SBATCH --mail-user=you@email   # Set your e-mail address
#SBATCH --account=x12345        # Charge resources on this
                                # project account

# execute serial programs, e.g.
cdo <operator> <ifile> <ofile>
```

Note: The shared partition has a limit of 1280MB memory per CPU. In case your serial job needs more memory you have to increase the number of tasks (using option `--ntasks`) although you might not use all these CPUs.

OpenMP job without HyperThreading

```
#!/bin/bash

#SBATCH --job-name=my_job       # Specify job name
#SBATCH --partition=shared      # Specify partition name
#SBATCH --ntasks=1              # Specify max. number of tasks
                                # to be invoked
#SBATCH --cpus-per-task=16      # Specify number of CPUs per task
#SBATCH --time=00:30:00         # Set a limit on the total run time
#SBATCH --account=x12345        # Charge resources on this
                                # project account

# bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=verbose,granularity=core,compact,1
export KMP_STACKSIZE=64M

# execute OpenMP programs, e.g.
cdo -P 8 <operator> <ifile> <ofile>
```

Note: You need to specify the value of `--cpus-per-task` as multiple of Hyper-Threads (HT). The environment variable `KMP_AFFINITY` needs to be set correspondingly. Whether HT is used or not is defined via the envVar `KMP_AFFINITY`, see 4.4.2 for details.

OpenMP job with HyperThreading

```
#!/bin/bash
```

```
#SBATCH --job-name=my_job       # Specify job name
#SBATCH --partition=shared      # Specify partition name
#SBATCH --ntasks=1              # Specify max. number of tasks
                                # to be invoked
#SBATCH --cpus-per-task=8       # Specify number of CPUs per task
#SBATCH --time=00:30:00         # Set a limit on the total run time
#SBATCH --account=x12345        # Charge resources on this
                                # project account

# bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=verbose,granularity=thread,compact,1
export KMP_STACKSIZE=64M

# execute OpenMP programs, e.g.
cdo -P 8 <operator> <ifile> <ofile>
```

MPI job without HyperThreading

The overall setting of the batch script does not vary whether one is using Intel MPI or bullx MPI (or any other MPI implementation). Only specific modules might be used and/or environmental variables should be set in order to fine-tune the used MPI. Especially, the parallel application should always be started using the `srun` command instead of invoking mpirun, mpiexec or others.

The following example allocates 8 full nodes and uses 24 physical cores per node. The total number of tasks is 192.

```
#!/bin/bash
#SBATCH --job-name=my_job       # Specify job name
#SBATCH --partition=compute     # Specify partition name
#SBATCH --nodes=8               # Specify number of nodes
#SBATCH --ntasks-per-node=24    # Specify number of tasks on each node
#SBATCH --time=00:30:00         # Set a limit on the total run time
#SBATCH --mail-type=FAIL        # Notify user by email
#SBATCH --mail-user=your@email # Set your e-mail address
#SBATCH --account=xz0123        # Charge resources on project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

# Environment settings to run a MPI parallel program compiled with
# Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
module load fca/2.5.2379
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR
```

```
# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Environment settings to run a MPI parallel program compiled with
# Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs
# compiled with any MPI library
srun -l --cpu_bind=cores --distribution=block:cyclic ./myprog
```

MPI job with HyperThreading

The following example allocates 4 full nodes and uses 48 logical CPUs per node. The total number of tasks is 192.

```
#!/bin/bash
#SBATCH --job-name=my_job       # Specify job name
#SBATCH --partition=compute     # Specify partition name
#SBATCH --nodes=4               # Specify number of nodes
#SBATCH --ntasks-per-node=48    # Specify number of tasks on each node
#SBATCH --time=00:30:00         # Set a limit on the total run time
#SBATCH --mail-type=FAIL        # Notify user by email
#SBATCH --mail-user=your@email # Set your e-mail address
#SBATCH --account=xz0123        # Charge resources on project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

# Environment settings to run a MPI parallel program compiled with
# Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
module load fca/2.5.2379
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Environment settings to run a MPI parallel program compiled with
# Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs
```

```
# compiled with any MPI library
srun -l --cpu_bind=threads --distribution=block:cyclic ./myprog
```

Instead of specifying the choice to use HyperThreads or not explicitly via `--cpus-per-task` and `--cpu_bind`, one might also use the srun option `--hint=[no]multithread`. The following example allocates one full node and uses 24 tasks without HyperThreads for the first program run and then 48 tasks using HyperThreads for the second run. Such a procedure might be used in order to see whether an application takes benefits of the use of HyperThreads or not.

```
#!/bin/bash
#SBATCH --job-name=my_job     # Specify job name
#SBATCH --partition=compute   # Specify partition name
#SBATCH --nodes=1             # Specify number of nodes
#SBATCH --time=00:30:00       # Set a limit on the total run time
#SBATCH --account=x12345      # Charge resources on this
                              # project account

# Environment settings to execute a parallel program compiled
# with Intel MPI
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
export I_MPI_FABRICS=shm:dapl
export I_MPI_FALLBACK=0
export I_MPI_DAPL_UD=enable

# First check how myprog performs without Hyper-Threads
srun -l --cpu_bind=verbose --hint=nomultithread --ntasks=24 ./myprog

# Second check how myprog performs with Hyper-Threads
srun -l --cpu_bind=verbose --hint=multithread --ntasks=48 ./myprog
```

Hybrid MPI/OpenMP job without Hyper-Threading

The following job example will allocate 4 compute nodes for 1 hour. The job will launch 24 MPI tasks in total, 6 tasks per node and 4 OpenMP threads per task. On each node 24 cores will be used.

```
#!/bin/bash
#SBATCH --job-name=my_job       # job name
#SBATCH --partition=compute     # partition name
#SBATCH --nodes=4               # number of nodes
#SBATCH --ntasks-per-node=6     # number of (MPI) tasks per node
#SBATCH --time=01:00:00         # Set a limit on the total run time
#SBATCH --mail-type=FAIL        # Notify user by email
#SBATCH --mail-user=your@email # Set your e-mail address
#SBATCH --account=xz0123        # Charge resources on project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

# Bind your OpenMP threads
export OMP_NUM_THREADS=4
export KMP_AFFINITY=verbose,granularity=core,compact,1
export KMP_STACKSIZE=64m
```

```
# Environment settings to run a MPI/OpenMP parallel program compiled
# with Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
module load fca/2.5.2379
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs compiled
# with any MPI library
srun -l --cpu_bind=cores --cpus-per-task=8 ./myprog
```

Hybrid MPI/OpenMP job with Hyper-Threading

The following example will run on 2 compute nodes having 6 MPI tasks per node and starting 8 threads per node using Hyper-Threading.

```
#!/bin/bash
#SBATCH --job-name=my_job       # job name
#SBATCH --partition=compute     # partition name
#SBATCH --nodes=2               # number of nodes
#SBATCH --ntasks-per-node=6     # number of (MPI) tasks on each node
#SBATCH --time=01:00:00         # Set a limit on the total run time
#SBATCH --mail-type=FAIL        # Notify user by email
#SBATCH --mail-user=your@email # Set your e-mail address
#SBATCH --account=xz0123        # Charge resources on project account
#SBATCH --output=my_job.o%j     # File name for standard output
#SBATCH --error=my_job.e%j      # File name for standard error output

# Bind your OpenMP threads
export OMP_NUM_THREADS=8
export KMP_AFFINITY=verbose,granularity=thread,compact,1
export KMP_STACKSIZE=64m

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Bullx MPI and Mellanox libraries, load environment
module load intel
module load mxm/3.3.3002
```

```
module load fca/2.5.2379
module load bullxmpi_mlx/bullxmpi_mlx-1.2.8.3

# Settings for Open MPI and MXM (MellanoX Messaging) library
export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

# Disable GHC algorithm for collective communication
export OMPI_MCA_coll=^ghc

# Environment settings to run a MPI/OpenMP parallel program compiled
# with Intel MPI, load environment
module load intelmpi
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

# Use srun (not mpirun or mpiexec) command to launch programs
# compiled with any MPI library
srun -l --cpu_bind=cores --cpus-per-task=8 ./myprog
```

4.4 Advanced SLURM Features

4.4.1 Hyper-Threading (HT)

Similar to the IBM Power6 used in BLIZZARD, the Haswell processors deployed for MISTRAL offer the possibility of Simultaneous Multithreading (SMT) in the form of the Intel Hyper-Threading (HT) Technology. With HT enabled each (physical) processor core can execute two threads or tasks simultaneously. The operating system thus lists a total of 48 logical cpus or Hardware Threads (HWT). Therefore, a maximum of 48 processes can be executed on each compute node without overbooking.

Each compute node on MISTRAL consists of two Intel Xeon E5-2680 v3 processors, located on socket zero and one, with 12 physical cores each. These cores are numbered 0 to 23 and the hardware threads are numbered 24 to 47. Figure 4.1 depicts a node schematically and illustrates the naming convention.

On MISTRAL we have HT enabled on each compute node and SLURM always uses the option `--threads-per-core=2` implicitly, such that the user is urged to bind the tasks/threads in an appropriate way. In Section 4.3 there are examples (commands and job scripts) on how to use HT or not.

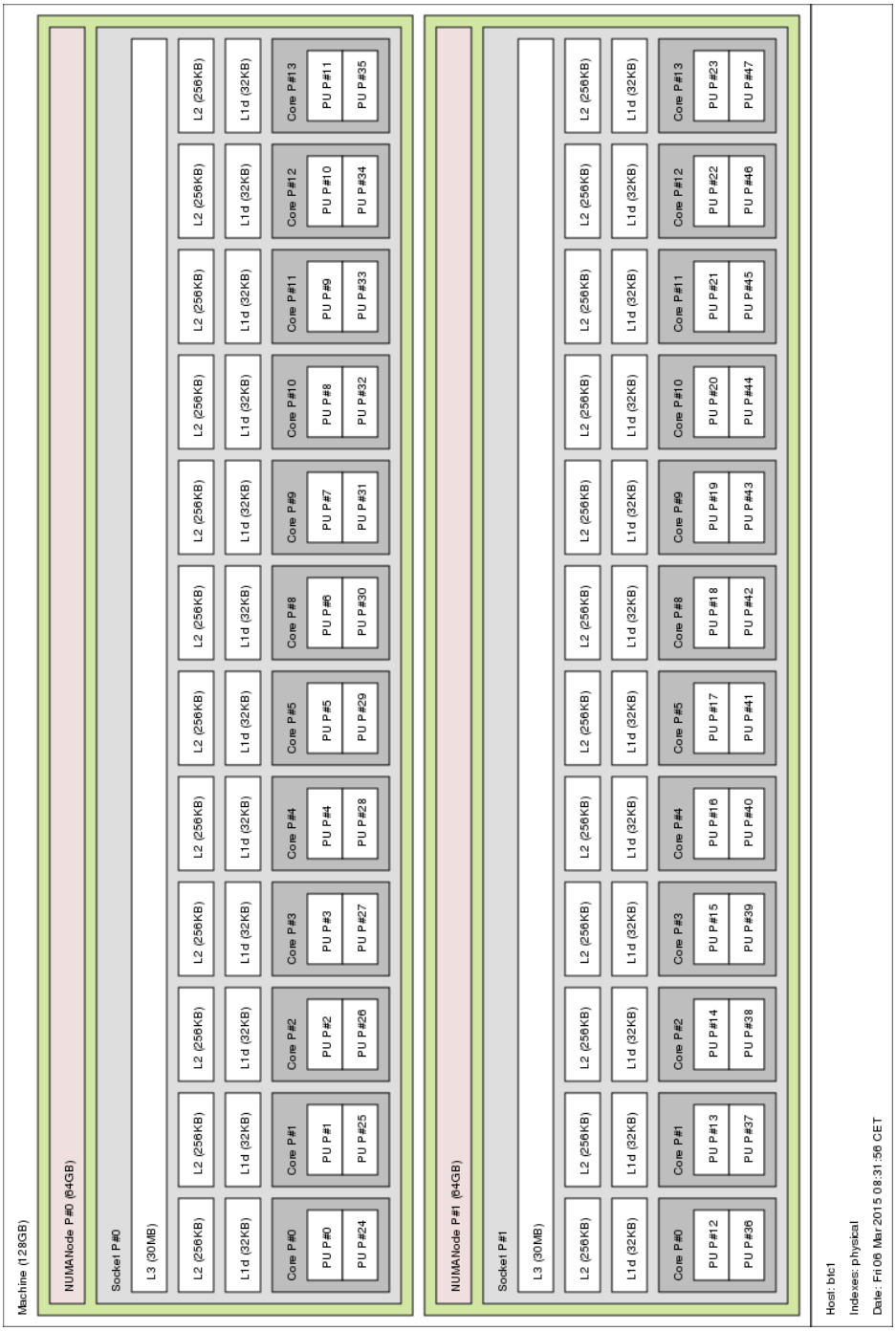

Figure 4.1: Schematic illustration of compute nodes

4.4.2 Process and Thread Binding

OpenMP jobs

Thread binding is done via Intel runtime library using the `KMP_AFFINITY` environment variable. The syntax is

KMP_AFFINITY=[<modifier>,...]<**type**>[,<permute>][,<offset>]

with

- modifier
  - verbose: giving detailed output on how binding was done

  - granularity=core: reserve full physical cores (i.e. two logical CPUs) to run threads on
  - granularity=thread/fine: reserve logical CPUs / HyperThreads to run threads
- type
  - compact: places the threads as close to each other as possible
  - scatter: distributes the threads as evenly as possible across the entire allocation
- permute: controls which levels are most significant when sorting the machine topology map, i.e.. 0=CPUs (default), 1=cores, 2=sockets/LLC
- offset: indicates the starting position for thread assignment

For details please take a look at the Intel manuals or contact DKRZ user's consultancy. In most cases use

```
export KMP_AFFINITY=granularity=core,compact,1
```

if you do **not** want to use HyperThreads and

```
export KMP_AFFINITY=granularity=thread,compact,1
```

if you intend to use HyperThreads. You might also try scatter instead of compact placement to take benefit from bigger L3 cache.

MPI jobs

Process/task binding can be done via srun options `--cpu_bind` and `--distribution`. The syntax is

```
--cpu_bind=[{quiet,verbose},]type
--distribution=<block|cyclic|arbitrary|plane=<options>[:block|cyclic]>
```

with

- type:
  - cores: bind to physical cores
  - threads: bind to logical CPUs / HyperThreads
- first distribution method (before the ":") controls the distribution of resources across nodes
- second (optional) distribution method (after the ":") controls the distribution of resources across sockets within a node

For details please take a look at the manpage of srun or contact DKRZ user's consultancy. In most cases use

```
srun --cpu_bind=verbose,cores --distribution=block:cyclic ./myapp
```

if you do not want to use HyperThreads and

```
srun --cpu_bind=verbose,threads --distribution=block:cyclic ./myapp
```

if you intend to use HyperThreads. You might also benefit from different task distributions than block:cyclic.

Hybrid MPI/OpenMP jobs

In this case you need to combine the two binding methods mentioned above. Keep in mind that we are using `--threads-per-core=2` throughout the cluster. Hence you need to specify the amount of CPUs per process/task on the basis of HyperThreads even if you do not intend to use HyperThreads! The following table gives an overview on how to achieve correct binding using a full node

	MPI intranode distribution of tasks =	
	srun –distribution=block:block	**srun –distribution=block:cyclic**

no OpenMP, no HT	`#SBATCH --tasks-per-node=24`
	`srun --cpu_bind=cores`
	
	task0:cpu{0,24}, task1:cpu{1,25}, ...
		`#SBATCH --tasks-per-node=24`
		`srun --cpu_bind=cores`
		
		task0:cpu{0,24}, task1:cpu{12,36}, ...

no OpenMP, HT	`#SBATCH --tasks-per-node=48`
	`srun --cpu_bind=threads`
	
	task0:cpu0, task1:cpu24, task2:cpu1, ...
		`#SBATCH --tasks-per-node=48`
		`srun --cpu_bind=threads`
		
		task0:cpu0, task1:cpu12, task2:cpu1, ...

4 OpenMP threads, no HT	`#SBATCH --tasks-per-node=6`
	`export OMP_NUM_THREADS=4`
	`export KMP_AFFINITY=\`
	`granularity=core,\`
	`compact,1`
	`srun --cpu_bind=cores \`
	`--cpus-per-task=8`
	
	task0:cpu{0,1,2,3,24,25,26,27},
	task1:cpu{4,5,6,7,28,29,30,31},
	...
	task0-thread0:cpu{0,24},
	task0-thread1:cpu{1,25},...
		`#SBATCH --tasks-per-node=6`
		`export OMP_NUM_THREADS=4`
		`export KMP_AFFINITY=\`
		`granularity=core,\`
		`compact,1`
		`srun --cpu_bind=cores \`
		`--cpus-per-task=8`
		
		task0:cpu{0,1,2,3,24,25,26,27},
		task1:cpu{12,13,14,15,36,37,38,39},
		...
		task0-thread0:cpu{0,24},
		task0-thread1:cpu{1,25},...

4 OpenMP threads, HT	`#SBATCH --tasks-per-node=12`
	`export OMP_NUM_THREADS=4`
	`export KMP_AFFINITY=\`
	`granularity=tread,\`
	`compact,1`
	`srun --cpu_bind=threads \`
	`--cpus-per-task=4`
	
	task0:cpu{0,1,24,25},
	task1:cpu{2,3,26,27},
	...
	task0-thread0:cpu0,
	task0-thread1:cpu1,
	task0-thread2:cpu24,...
		`#SBATCH --tasks-per-node=12`
		`export OMP_NUM_THREADS=4`
		`export KMP_AFFINITY=\`
		`granularity=thread,\`
		`compact,1`
		`srun --cpu_bind=threads \`
		`--cpus-per-task=4`
		
		task0:cpu{0,1,24,25},
		task1:cpu{12,13,36,37},
		...
		task0-thread0:cpu0,
		task0-thread1:cpu1,
		task0-thread2:cpu24,...

4.4.3 MPMD

SLURM supports the MPMD (Multiple Program Multiple Data) execution model that can be used for MPI applications, where multiple executables can have one common `MPI_COMM_WORLD` communicator. In order to use MPMD the user has to set the srun option `--multi-prog <filename>`. This option expects a configuration text file as an argument, in contrast to the SPMD (Single Program Multiple Data) case where srun has to be given the executable.

Each line of the configuration file can have two or three possible fields separated by space and the format is

```
<list of task ranks> <executable> [<possible arguments>]
```

In the first field a comma separated list of ranks for the MPI tasks that will be spawned is defined. Possible values are integer numbers or ranges of numbers. The second field is the path/name of the executable. And the third field is optional and defines the arguments of the program.

Example

Listing 4.1: Jobscript frame for the coupled MPI-ESM model using 8 nodes

```
#!/bin/bash

#SBATCH --nodes=8
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00
#SBATCH --exclusive
#SBATCH --account=x12345

# Atmosphere
ECHAM_NPROCA=6
ECHAM_NPROCB=16

# Ocean
MPIOM_NPROCX=12
MPIOM_NPROCY=8

# Paths to executables
ECHAM_EXECUTABLE=../bin/echam6
MPIOM_EXECUTABLE=../bin/mpiom.x

# Derived values useful for running
(( ECHAM_NCPU = ECHAM_NPROCA * ECHAM_NPROCB ))
(( MPIOM_NCPU = MPIOM_NPROCX * MPIOM_NPROCY ))
(( NCPU = ECHAM_NCPU + MPIOM_NCPU ))
(( MPIOM_LAST_CPU = MPIOM_NCPU - 1 ))
(( ECHAM_LAST_CPU = NCPU - 1 ))

# create MPMD configuration file
cat > mpmd.conf <<EOF
0-${MPIOM_LAST_CPU} $MPIOM_EXECUTABLE
${MPIOM_NCPU}-${ECHAM_LAST_CPU} $ECHAM_EXECUTABLE
EOF
```

```
# Run MPMP parallel program using Intel MPI
module load intelmpi

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
export I_MPI_FABRICS=shm:dapl
export I_MPI_FALLBACK=0
export I_MPI_DAPL_UD=enable

srun -l --cpu_bind=verbose,cores --multi-prog mpmd.conf
```

4.4.4 Frequency Scaling

The Intel Haswell processor allows for CPU frequency scaling which in general enables the operating system to scale the CPU frequency up or down in order to save power. CPU frequencies can be scaled automatically depending on the system load or manually by userspace programs. This is done via power schemes for the CPU - so called governors. Only one may be active at a time. The default governor is "ondemand" which allows the operating system to scale down the CPU frequency on the compute nodes to 1.2GHz if they are in idle state. The user can set the governor to "userspace" in order to allow for different CPU frequencies. Therefore the batch job needs to define the desired behaviour via the environmental variable `SLURM_CPU_FREQ_REQ` or via the `srun` option `--cpu-freq`.

To set a fixed frequency of 2.5GHz (2500000 kHz) use:

```
export SLURM_CPU_FREQ_REQ=2500000
```

Other allowed frequencies are 1.2, 1.3,..., 2.5 GHz.

To enable automatic frequency scaling depending on the workload use:

```
export SLURM_CPU_FREQ_REQ=ondemand
```

By default `srun` configures all CPUs to run at fixed frequency of 2.5GHz in order to get similar wallclock runtime between different jobs if no options (or binaries) are changed.

4.4.5 Job Steps

Job steps can be thought of as small allocations or jobs inside the current job/allocation. Each call of srun creates a job-step which implies that one job/allocation given via sbatch can have one or several job steps executed in parallel or sequentially. Instead of submitting many single-node jobs, the user might also use job steps inside a single job having multiple nodes allocated. A job using job steps will be accounted for all the nodes of the allocation regardless if all nodes are used for job steps or not.

The following example uses job steps to execute MPI programs in different job steps sequentially after each other and also parallel to each other inside the same job allocation. In total 4 nodes are allocated: the first 2 job steps run on all nodes after each other, while the job steps 3 and 4 run in parallel each using only 2 nodes.

```
#!/bin/bash

#SBATCH --nodes=4
#SBATCH --time=00:30:00
#SBATCH --account=x12345
```

```
# run 2 job steps after each other
srun -N4 --ntasks-per-node=24 --time=00:10:00 ./mpi_prog1
srun -N4 --ntasks-per-node=24 --time=00:20:00 ./mpi_prog2

# run 2 job steps in parallel
srun -N1 -n24 ./mpi_prog3 &
srun -N3 --ntasks-per-node=24 ./mpi_prog4 &
```

4.4.6 Dependency Chains

SLURM supports dependency chains which are collections of batch jobs with defined dependencies. Job dependencies can be defined using the `--dependency` argument of sbatch.

```
#!/bin/bash

#SBATCH --dependency=<type>
```

The available dependency types for job chains are

- **after:<jobID>** job starts when job with <jobID> begun execution
- **afterany:<jobID>** job starts when job with <jobID> terminates
- **afterok:<jobID>** job starts when job with <jobID> terminates successfully
- **afternotok:<jobID>** job starts when job with <jobID> terminates with failure
- **singleton** jobs starts when any previously job with the same job name and user terminates

4.4.7 Job Arrays

SLURM supports job arrays which is a mechanism for submitting and managing collections of similar jobs quickly and easily. Job arrays are only supported for the `sbatch` command and are defined using the option `--array=<indices>`. All jobs use the same initial options (e.g. number of nodes, time limit, etc.), however since each part of the job array has access to the `SLURM_ARRAY_TASK_ID` environment variable individual setting for each job is possible. For example the following job submission

```
bash$ sbatch --array=1-3 -N1 slurm_job_script.sh
```

will generate a job array containing three jobs. Assuming that the jobID reported by sbatch is 42, then the parts of the array will have the following environment variables set:

```
# array index 1
SLURM_JOBID=42
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=1

# array index 2
SLURM_JOBID=43
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=2
```

```
# array index 3
SLURM_JOBID=44
SLURM_ARRAY_JOB_ID=42
SLURM_ARRAY_TASK_ID=3
```

Some additional options are available to specify the *stdin*, *stdout*, and *stderr* file names: option `%A` will be replaced by the value of `SLURM_ARRAY_JOB_ID` and option `%a` will be replaced by the value of `SLURM_ARRAY_TASK_ID`.

The following example creates a job array of 42 jobs with indices 0-41. Each job will run on a separate node with 24 tasks per node. Depending on the queuing situation, some jobs may be running and some may be waiting in the queue. Each part of the job array will execute the same binary but with different input files.

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --output=prog-%A_%a.out
#SBATCH --error=prog-%A_%a.err
#SBATCH --time=00:30:00
#SBATCH --array=0-41
#SBATCH --account=x12345

srun --ntasks-per-node=24 ./prog input_${SLURM_ARRAY_TASK_ID}.txt
```

4.5 SLURM Command Examples

4.5.1 Query Commands

Normally, the jobs will pass through several states during their life-cycle. Typical job states from submission until completion are: PENDING (PD), RUNNING (R), COMPLETING (CG) and COMPLETED (CD). However there are plenty of possible job states for SLURM. The following describes the most common states:

CA CANCELLED : Job was explicitly cancelled by the user or an administrator. The job may or may not have been initiated.

CD COMPLETED : Job has terminated all processes on all nodes.

CF CONFIGURING : Job has been allocated resources, but is waiting for them to become ready for use.

CG COMPLETING : Job is in the process of completing. Some processes on some nodes may still be active.

F FAILED : Job terminated with non-zero exit code or other failure condition.

NF NODE_FAIL : Job terminated due to failure of one or more allocated nodes.

PD PENDING : Job is awaiting resource allocation.

R RUNNING : Job currently has an allocation.

TO TIMEOUT : Job terminated upon reaching its walltime limit.

Some examples how users can query their jobs status are.

```
bash$ squeue
JOBID PARTITION     NAME     USER ST TIME NODES NODELIST(REASON)
13194   compute MR_2_01P  k203059 PD 0:00     13 (PartitionTimeLimit)
13263   compute LR0014.r  k208024  R 4:03     16 m[10002-10017]
```

Check the Queue for one user only:

```
bash$ squeue -u $USER
JOBID PARTITION     NAME     USER ST TIME NODES NODELIST(REASON)
13263   compute LR0014.r  k208024  R 4:03     16 m[10002-10017]
```

Depending on the options, the `sinfo` command will print the states of the partitions and the nodes. The partitions may be in state UP, DOWN or INACTIVE. The UP state means that a partition will accept new submissions and the jobs will be scheduled. The DOWN state allows submissions to a partition but the jobs will not be scheduled. The INACTIVE state means that not submissions are allowed.

```
bash$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up    8:00:00     31  maint m[11440-11470]
compute*     up    8:00:00     13  down* m[10000,10278,10286,10438,
...
compute*     up    8:00:00    812   idle m[10001-10034,10036-10041,
...
gpu          up    4:00:00      1  down* mg102
gpu          up    4:00:00     11   idle mg[100-101,103-111]
```

The nodes also can be in various states. Node state code may be shortened according to the size of the printed field. The following shows the most common node states:

alloc ALLOCATED : The node has been allocated.

comp COMPLETING : The job associated with this node is in the state of COMPLETING.

down DOWN : The node is unavailable for use.

drain DRAINING , DRAINED : While in DRAINING state any running job on the node will be allowed to run until completion. After that and in DRAIN state the node will be unavailable for use.

idle IDLE : The node is not allocated to any jobs and is available for use.

maint MAINT : The node is currently in a reservation with a flag of maintenance.

resv RESERVED : The node is in an advanced reservation and not generally available.

A listing based in nodes can be viewed as follows:

```
bash$ sinfo -N
NODELIST                                                NODES PARTITION STATE
m[10000,10278,10286,10438,10498,10518,10554,               13  compute* down*
m10000                                                      1    shared down*
m[10001-10017,10036-10049,11296-11313,                     63    shared idle
m[10001-10017,10036-10053,10072-10107,                   1318  compute* idle
m[10018-10035,11314-11331]                                 36    shared alloc
m[10018-10035,10054-10071,10108-10110,10113-              165  compute* alloc
m[11512-11517,11519-11531,11533-11553,11555-               45   prepost idle
m[11518,11532,11554]                                        3   prepost drain*
mg[100-101,103-111]                                        11       gpu idle
mg102                                                       1       gpu down*
```

Query configuration and limits for one specific partition (here compute):

```
bash$ scontrol show partition compute
```

Check one node (here m10010):

```
bash$ scontrol show node m10010
```

4.5.2 Job Control

The `scontrol` command is primarily used by the administrators to manage SLURM's configuration. However it provides also some functionality for the users to manage jobs and get some information about the system configuration.

Show information about the job 4242:

```
bash$ scontrol show job 4242
```

Hold a job:

```
bash$ scontrol hold 4242
bash$ squeue
 JOBID PARTITION     NAME     USER ST   TIME  NODES NODELIST(REASON)
  4242    nightly tst_job  b123456 PD   0:00      1 (JobHeldUser)
```

Release a job:

```
bash$ scontrol release 4242
bash$ squeue
 JOBID PARTITION     NAME     USER ST   TIME  NODES NODELIST(REASON)
  4242    nightly tst_job  b123456  R   0:01      1 m[10007-10011]
```

With `scancel`, we can signal or cancel jobs, job arrays or job steps. Cancel a specific job:

```
bash$ scancel 4711
```

Cancel all jobs in queue (pending, running, etc.) from user x123456 in interactive mode (user must confirm for each operation):

```
bash$ scancel --interactive -u x123456
```

With `sstat`, one can get various status information about running job-steps, for example minimum, maximum and average values for metrics like CPU time, Virtual Memory (VM) usage, Resident Set Size (RSS), Disk I/O, Tasks number, etc.

Display default status information for job 4242:

```
bash$ sstat -j 4242
```

Display the defined metrics for job 4242 in parsable format:

```
bash$ sstat -P --format=JobID,AveCPU,AvePages,AveRSS,AveVMSize -j 4242
```

4.5.3 Accounting Commands

With `sacct` one can get accounting information and data for the jobs and jobsteps that are stored in SLURM's accounting database. SLURM stores the history of all jobs in the database but each user has permissions to check only his/her own jobs.

Show job information in long format for default period (starting from 00:00 today until now):

```
bash$ sacct -l
```

Show job only information (without jobsteps) starting from the defined date until now:

```
bash$ sacct -S 2015-01-07T00:42:00 -X
```

Show job information with different format and specified time frame:

```
bash$ sacct -X -u b123456 --format="jobid,nnodes,nodelist,state,exit"
        -S 2015-01-01 -E 2015-31-01T23:59:59
```

The `sacctmgr` command is mainly used by the administrators to view or modify accounting information and data in the accounting database. This command provides also an interface with limited permissions to the users for some querying actions. The most useful command is to show all associations a user is allowed to submit jobs:

```
bash$ sacctmgr show assoc where user=<user_id>
```

List all or the specified QoS:

```
bash$ sacctmgr show qos [where name=<qos_name>]
```

Show the privileges of my user:

```
bash$ sacctmgr show user
```

Show cluster information:

```
bash$ sacctmgr show cluster
```